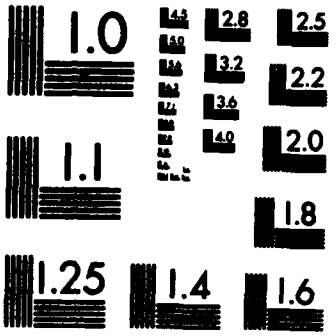
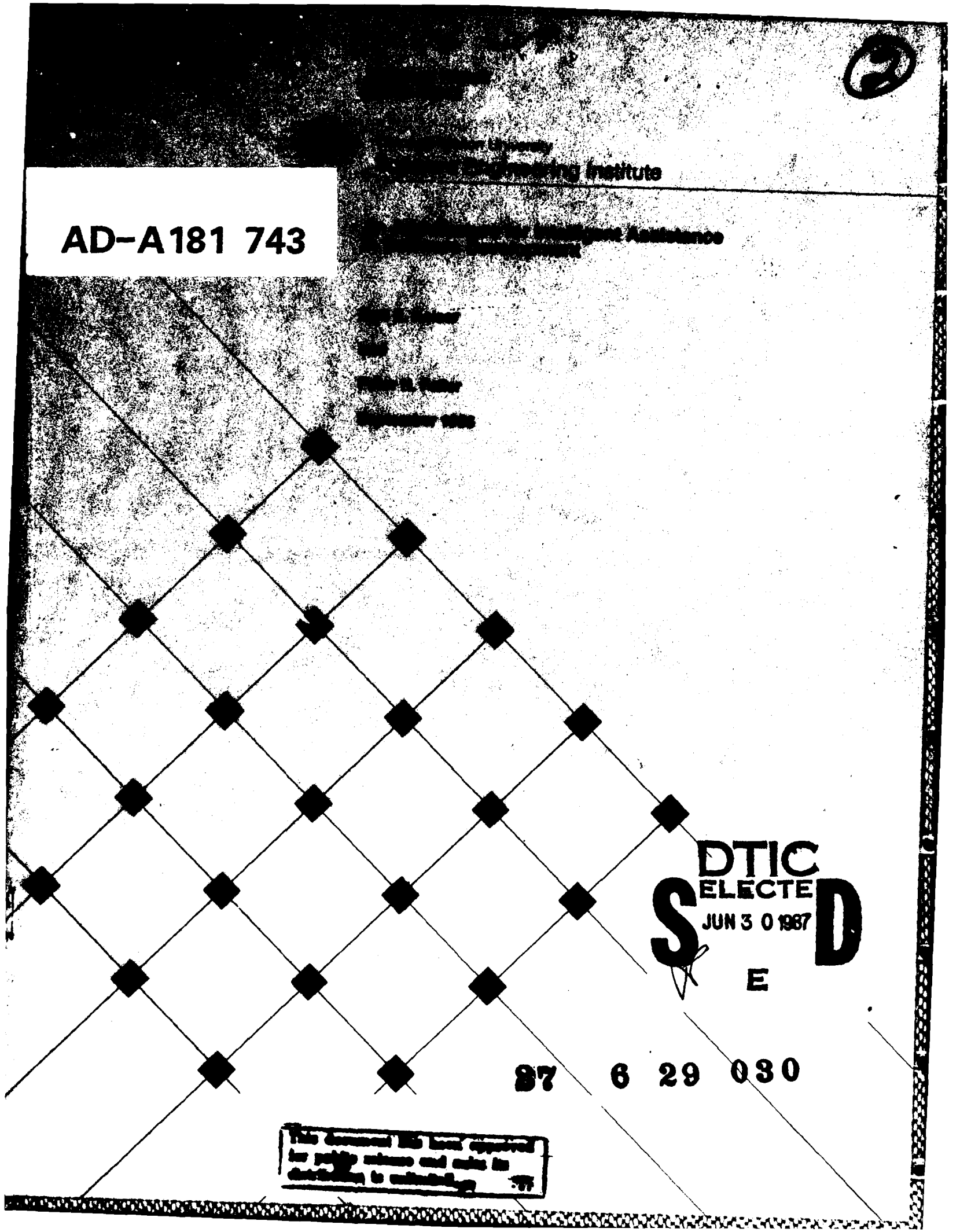
AD-A181 743 AN ARCHITECTURE FOR INTELLIGENT ASSISTANCE IN SOFTWARE DEVELOPMENT..(U) CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.. G E KAISER ET AL. SEP 86 1/1

UNCLASSIFIED CMU/SEI-86-TM-12 ESD-TR-86-219 F/G 12/5 NL





SECURITY CLASSIFICATION OF THIS PAGE

A181 743

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION UNLIMITED, UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT UNCLASSIFIED, UNLIMITED, DTIC, NTIS
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) SEI-86-TM-12	5. MONITORING ORGANIZATION REPORT NUMBER(S) ESD-TR-86-219
6a. NAME OF PERFORMING ORGANIZATION SOFTWARE ENGINEERING INST. / 6b. OFFICE SYMBOL (If applicable) SEI	7a. NAME OF MONITORING ORGANIZATION SEI JOINT PROGRAM OFFICE
6c. ADDRESS (City, State and ZIP Code) CARENGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	7b. ADDRESS (City, State and ZIP Code) ESD/XRS1 HANSCOM AIR FORCE BASE HANSCOM, MA 01731
8a. NAME OF FUNDING/SPONSORING ORGANIZATION SEI JPO / 8b. OFFICE SYMBOL (If applicable) ESD/XRS1	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F19628 85 C 0003
8c. ADDRESS (City, State and ZIP Code) CARNEGIE-MELLON UNIVERSITY PITTSBURGH, PA 15213	10. SOURCE OF FUNDING NOS. PROGRAM ELEMENT NO.: 63752F / PROJECT NO.: N/A / TASK NO.: N/A / WORK UNIT NO.: N/A
11. TITLE (Include Security Classification) AN ARCHITECTURE FOR INTELLIGENT ASSISTANCE IN SOFTWARE DEVELOPMENT	
12. PERSONAL AUTHOR(S) GAIL KAISER AND PETER FEILER	
13a. TYPE OF REPORT FINAL	13b. TIME COVERED FROM ... TO ...
14. DATE OF REPORT (Yr., Mo., Day) SEPTEMBER 86	15. PAGE COUNT 16
16. SUPPLEMENTARY NOTATION N/A	

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB. GR.	

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

WE DEFINE AN ARCHITECTURE FOR A SOFTWARE ENGINEERING ENVIRONMENT THAT BEHAVES AS AN INTELLIGENT ASSISTANT. OUR ARCHITECTURE CONSISTS OF TWO KEY ASPECTS, AN OBJECTBASE AND A MODEL OF THE SOFTWARE DEVELOPMENT PROCESS. OUR OBJECTBASE IS ADAPTED FROM OTHER RESEARCH, BUT OUR MODEL IS UNIQUE IN THAT IT CONSISTS PRIMARILY OF RULES THAT DEFINE THE PRECONDITIONS AND MULTIPLE POSTCONDITIONS OF SOFTWARE DEVELOPMENT TOOLS. METARULES DEFINE FORWARD AND BACKWARD CHAINING AMONG THE RULES. OUR MOST SIGNIFICANT CONTRIBUTION TO OPPORTUNISTIC PROCESSING, WHEREBY THE ENVIRONMENT AUTOMATICALLY PERFORMS SOFTWARE DEVELOPMENT ACTIVITIES AT SOME TIME BETWEEN WHEN THEIR PRECONDITIONS ARE SATISFIED AND WHEN THEIR POSTCONDITIONS ARE REQUIRED. FURTHER, OUR MODEL DEFINES STRATEGIES THAT GUIDE THE ASSISTANT IN CHOOSING AN APPROPRIATE POINT FOR CARRYING OUT EACH ACTIVITY.

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED ☒ SAME AS RPT. ☒ DTIC USERS ☒	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED, UNLIMITED, DTIC, NTIS
22a. NAME OF RESPONSIBLE INDIVIDUAL KARL H. SHINGLER	22b. TELEPHONE NUMBER (Include Area Code) 412 268-7630 / 22c. OFFICE SYMBOL SEI JPO

DD FORM 1473, 83 APR EDITION OF 1 JAN 73 IS OBSOLETE.

SECURITY CLASSIFICATION OF THIS PAGE

Technical Memorandum
SEI-86-TM-12
September 1986

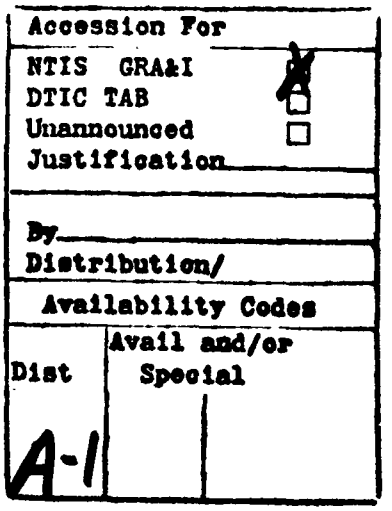
An Architecture for Intelligent Assistance in Software Development

by

Gail E. Kaiser*
Columbia University

and

Peter H. Feiler
Software Engineering Institute



*This paper was written while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

This work was sponsored by the Department of Defense.

Table of Contents

List of Figures

An Architecture for Intelligent Assistance in Software Development

Gail E. Kaiser[1], Peter H. Feiler

ABSTRACT: We define an architecture for a software engineering environment that behaves as an intelligent assistant. Our architecture consists of two key aspects, an objectbase and a model of the software development process. Our objectbase is adapted from other research, but our model is unique in that is consists primarily of rules that define the preconditions and multiple postconditions of software development tools. Metarules define forward and backward chaining among the rules. Our most significant contribution is *opportunistic processing*, whereby the environment automatically performs software development activities at some time between when their preconditions are satisfied and when their postconditions are required. Further, our model defines strategies that guide the assistant in choosing an appropriate point for carrying out each activity.

1 Introduction

In 1973, Winograd [30] presented his dream of an intelligent assistant for programmers that would understand what it does: it would be based on an explicit model of the programming world. Winograd described an imaginary programming environment, A, that would assist programmers by providing early error checking, by answering questions about the program and the interactions among program parts, by handling trivial programming problems, and by automating simple debugging tasks.

Artificial intelligence research has moved closer to achieving this dream by developing a *knowledge-based* approach to programming, which includes relationships among program units, both in the abstract and with respect to a particular target system. The Masterscope package of Interlisp [27] and the CommonLisp Framework (CLF) [5] maintain cross-referencing information to answer queries about interactions among program units. Also, CLF's knowledgebase understands the abstract relationships among program units. For example, CLF 'knows' that a system consists of modules and individual software objects (functions, variables, *etc.*) and that object classes have particular properties, such as a maintainer and whether it has been compiled.

A knowledge-based programming environment also includes the rules governing the software development process. For example, wide-spectrum languages such as V [23] and Gist [1] have been augmented with rules that aid the programmer in translating from higher- to lower-level specifications and from specifications to executable code. CMS [13] provides a formal representation of the software project model, time (for scheduling), and software development activities. For example, if a 'capability' is a desired feature of the target system, then it must be realized by a 'component' of the system; a 'task' must be defined to specify who is in charge of the component and when it is due. Genesis's Activity Manager [19] provides similar facilities.

---

[1]The research presented in this paper was conducted while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

While work in AI was progressing, researchers in traditional software were addressing their version of Winograd's dream. *Tools* were developed that automated certain aspects of the programming process. For example, Make [8] automatically rederives an executable system when part of the source code changes. SCCS [22] requires programmers to reserve modules for change, thus ensuring orderly software evolution. RCS [28] supports multiple versions of software objects and manages separate lines of development.

Collections of tools were integrated into interactive programming environments that support a particular programming language. The Synthesizer [26] combines language-oriented editing that prevents syntactic errors with immediate feedback about static semantic errors; it also permits programmers to interleave execution and debugging with editing. The Gandalf Prototype [16] added a module interconnection language with incremental, intermodule consistency checking to a C programming environment similar to the Synthesizer.

Unfortunately, the knowledge-based approach and the tools approach have progressed more-or-less independently. The individual tools incorporate a small bit of knowledge about a particular programming problem, but this knowledge cannot be augmented. The programming environments are hardcoded with a particular view of the software development process that defines the interaction between the programmers and the target system, but this knowledge is not available to the users of the environment. The knowledge-based environments are much more general, but represent only a fraction of human expertise about software development and maintenance.

Working within the tools approach, the members of the Gandalf project (including the authors) developed a distributed, multi-user software engineering environment called SMILE [24, 7], which is relatively close to achieving intelligent assistance. SMILE presents a 'fileless environment' to its users, answers queries, and automatically invokes various tools. However, SMILE's knowledge of software objects and the programming process is hardcoded into the environment.

Our experience with SMILE provided insights into the development of practical environments and convinced us that a generalization of SMILE's internal architecture would aid in developing an intelligent assistant for software development and maintenance. Our architecture for intelligent assistance combines tools with knowledge. From the tools approach, we gain the years of experience of other computer scientists building and using particular tools and environments. From the knowledge-based approach, we gain a suitable structure for choosing among tools and automating the invocation of tools. Our architecture defines a basis for intelligent assistance that consists of two key aspects: an *objectbase* and a *model of the software development process*.

The objectbase maintains all software objects, including tools, and provides the environment with insight into the various classes of objects and the relationships among objects. For example, one object is a component of another, and a particular object may be applied to another object to produce a third.

The model imposes a structure on programming activities. It consists of an extensible collection of rules that specify the particular conditions that must exist for particular tools to be applied to

particular software objects. Metarules permit the environment to understand the rules and support *opportunistic processing*, where the environment performs activities when it knows the results of these activities will soon be required by its users. Opportunistic processing is the primary focus of this paper.

In this paper, we explain our architecture and how it meets certain fundamental requirements for supporting a software engineering environment that understands what it does. Section 2 presents the basis for intelligent assistance defined by our architecture. Section 3 describes how an intelligent assistant built on this framework can perform software development activities automatically to provide intelligent assistance to its users. Section 4 briefly describes our implementation.

2 A Basis for Intelligent Assistance

An intelligent assistant should understand what it is doing [30]. Most software tools are "moronic assistants" that know what they are doing, but do not understand the purpose of the objects they manipulate or how their tasks fit into the software development process. In other words, they may know the 'how' but not the 'why'.

For example, Make has a simplistic world model consisting of files and command lines. A 'makefile' defines dependencies among files and gives the command lines necessary for restoring consistency among dependent files. Make's notion of consistency is based on files and time: if the timestamp of an input file is later than the timestamp of an output file, then the indicated command line should be passed to the Unix™ shell. Make is used widely for generating a new executable version of a system after source files have been modified.

However, Make's 'knowledge' is primitive. Its objectbase consists of files that have a single attribute, their timestamp. Make knows nothing about applying tools to files: it just handles command lines as indivisible units. Make does not understand source files *vs.* object files, modules *vs.* systems, programmers or programming.

To give Make this knowledge we could define a notion of an *object*, which belongs to a *class*, such as 'system' or 'module'. Each class would define the *attributes*, or properties, of its objects. A 'module object code' object might have a 'history' attribute describing its generation and a 'derivation of' attribute pointing to the object representing the corresponding source code.

We could then define *rules* that model the part of the software development process relevant to Make. One rule might be that a 'programmer' object can modify a 'module' object; another rule might state that after such a modification, the 'module' object is no longer consistent with its 'derivation' attribute and there is an obligation to restore this consistency. A third rule might state that a precondition for a 'programmer' to test a 'system' is that all 'module object code' objects that are components of the corresponding 'executable system' must be consistent with their 'module'.

Given this knowledge, Make could be considered to be relatively more intelligent. Make would be easier to integrate with other tools that support configuration management, version control, task management, *etc.*, assuming these tools had similar knowledge of software objects and their roles in the development process.

We believe that an objectbase and a model of the development process are prerequisites to intelligent assistance. An assistant cannot understand why it performs particular activities unless it knows

- the properties of the objects it manipulates,
- the capabilities of certain objects (programmers and tools) to manipulate other objects,
- the preconditions required by each activity,
- the postconditions of each activity.

Therefore, our architecture specifies a general objectbase and an extensible collection of rules describing the preconditions and postconditions of activities, as well as hints and strategies that determine the degree of the environment's contributions. We briefly describe the objectbase here; the rules, hints and strategies are the topic of the following section.

Objectbase

We considered several possible forms for our objectbase. One possibility was the entity-relation-attribute model proposed for Genesis [20]. However, weaknesses of relational databases make them inadequate for software engineering environments [15]. To maximize flexibility, we chose an objectbase similar to those of object-oriented programming languages, such as Loops [25]. In particular, we adopted their support for multiple inheritance and active values. Unlike most such languages, however, we require a 'persistent' objectbase, one that retains its state across invocations of the environment. The same concepts are found in the objectbases supported by other knowledge-based environments, such as AP3 [2] and Refine™ [23].

In our objectbase, each object is an instance of a class, which defines certain attributes of each object and inherits other attributes from its superclass(es). Some attributes define the relationships among objects; others trigger activities when accessed and/or updated. The activities applicable to a class are defined as methods for the class.

This enables an intelligent assistant to expose its users only to the logical structure of the target software system. The environment consists of a set of typed, interconnected software objects representing the system and its history. The interconnections among software objects represent the logical structure of the system. Object types include module, procedure, type, design description, user task (or development step), user manual, *etc.* Typing permits the assistant to provide an object-oriented user interface similar to the Smalltalk-80™ environment [11], where the environment makes available to each user only those commands that are relevant to the object under consideration.

3 Opportunistic Processing

The objectbase also maintains the rules that model the software development process. These rules provide the meta-knowledge required for an environment to apply tools automatically. We call this behavior *opportunistic processing*, which offloads simple activities onto the intelligent assistant—menial activities, such as invoking the compiler and recording any errors found during compilation. This approach contrasts with some intelligent assistance systems, such as the Programmer's Apprentice (KBEmacs) [29] and CHI (previously PSI) [23], which focus on the separate problem of automatic programming.

3.1 Rules

We represent our model as a collection of rules similar to the production rules of Ops5 [4] in that each rule has a condition and action. When the condition is true, the action may be executed. Our rules differ from production rules in that the action is divided into two parts, an activity and a postcondition. Because our rules have postconditions, we refer to the original conditions as preconditions.

The *activity* part of a rule represents an integral software development task. For example, "compile module" is one activity and "change component" is another (a 'component' is a facility defined within a module, such as a procedure, a variable, a type, *etc.*). The specific editing commands applied during the course of the "change component" activity are not considered activities. "Fix bug" is not an activity, since it involves many tasks, perhaps involving several users. Thus our notion of an activity represents a middle-ground granularity.

Each activity is associated in the objectbase with a tool that performs the activity. One attribute of each tool is whether it can be invoked by the environment without human intervention. For example, the "compile module" activity is associated with the compiler, which can be applied by the intelligent assistant; the "edit component" activity is associated with a text editor (or a syntax-directed editor), which requires human interaction.

The *precondition* part of a rule — a boolean expression — must be true before an activity can be performed. The operands of a precondition include software objects and the attributes of software objects. For example, "notcompiled(module)" might be an appropriate precondition for the "compile module" activity. Another precondition for "compile module" would be "for all components c such that in(module, component c): analyzed(component c)", where "analyzed(c)" is true only if a static semantic analysis of component *c* finds no errors. An activity may have multiple preconditions that must be satisfied.

A *postcondition* becomes true when an activity is completed. Both preconditions and postconditions are written as well-formed formulas (wffs) in the first order predicate calculus. Our rules are based on Hoare's assertions [12], where a programming language construct is associated with its preconditions and postconditions; if the preconditions are true before the language construct is executed, then the postconditions will be true afterwards.

However, a programming activity may have multiple postconditions, exactly one of which is true after the activity terminates. Which of the various possibilities is true can be determined only by invoking the corresponding tool. For example, two postconditions for the "compile module" activity might be "compiled(module)" and "errors(module)". Here we follow the extension of Hoare's assertions proposed by Perry [18], where there must be multiple postconditions to represent the exceptional results of executing a procedure. This notion of postconditions distinguishes our architecture from CLF, Genesis' Activity Manager, and other expert systems that rely on condition/action rules. The most important advantages are that we can separate an activity from its results and therefore consider several alternative results within our model.

Two example rules are given in Figure 3.1. The first states the preconditions and the two postconditions for the "compile module" activity. The preconditions are given first, followed by the activity (within braces), followed by the postconditions. The alternative postconditions are separated by semicolons.

```
notcompiled(module) and
   for all components c such that in(module, component c):
       analyzed(component c);
    { compile module }
compiled(module);
errors(module);

equals(module, focus(userid)) and in(module, component);
    { edit component }
notanalyzed(component) and notcompiled(module);
```

Figure 1: Compile Rule and Edit Rule

3.2 Meta Rules

Our architecture supports the definition of metarules that guide the intelligent assistant's use of rules. One metarule states that if the preconditions of an activity are satisfied, and the activity can be performed by the assistant, then the assistant may perform the activity automatically. Consider the first rule in Figure 3.1. The metarule interprets this rule to mean that the assistant may compile any modules *M* such that all the components of *M* have been analyzed but *M* has not been compiled.

In this example, "notcompiled(module)" is one of the preconditions to the "compile module" activity; "errors(module)" is included as one of the possible postconditions. If the previous compilation failed, "errors(module)" will be true. The "compile module" activity cannot be performed when "errors(module)" is true, because its preconditions cannot be satisfied. If a user then edits a component, perhaps to fix the error, the second rule of Figure 3.1 states that "notcompiled(module)" will be set to true and the metarule permits compilation.

Importantly, this metarule states that the intelligent assistant may perform an activity when

preconditions are satisfied; it does not state that the assistant must perform the activity as soon as the preconditions are true, or at any time thereafter. However, the intelligent assistant may apply the tool and use *forward chaining* to determine additional activities whose preconditions are satisfied by the postconditions of the first activity. Therefore, we call this metarule the 'forward chaining metarule'.

Forward chaining supports behavior similar to language-oriented editors, such as the Synthesizer and Gnome [9]. When the user makes a subtree replacement in the abstract syntax tree representing the program, the editor automatically performs several actions. In the case of editors generated from attribute grammars [21], the editor automatically re-evaluates the values of attributes whose values may have changed. These attributes might represent the content of the symbol table and the object code for the program. Other editor generators automatically invoke action routines for type checking or code generation for modified program parts [6].

A second metarule states that if a user invokes a tool with unsatisfied preconditions, the intelligent assistant should use *backward chaining* to find activities it can perform whose postconditions might satisfy the preconditions of the activity requested by the user. In this case, the metarule states that the intelligent assistant must exhibit this behavior. We call this metarule the 'backward chaining metarule'.

Backward chaining supports behavior similar to Make, DSEE™ [14], Toolpack [17] and other software engineering tools in which a user may request regeneration of an executable system after changes have been made to its source code. The environment uses dependency information previously supplied by the software development team to determine which source files to recompile.

Sometimes our intelligent assistant attempts backwards chaining, but finds that the preconditions cannot be satisfied; in this case, the user is informed of the problem. The intelligent assistant is not expected to, for example, correct source code so that it will compile successfully. For example, our intelligent assistant might support a large team where multiple users should not change the same module simultaneously. Here, each user must reserve a module before changing it. The preconditions and postconditions for the "reserve module" activity are stated in the first rule shown in Figure 2 ("saved(module)" is true when the module has been saved by the version control tool), and the second rule states that the "change component" activity cannot be performed unless the module containing the component is reserved.

The "change component" activity permits the user to modify the specification of a component ("edit component" permits the user to modify only the body). The third rule of Figure 2 states that the containing module must be reserved along with any other modules that depend on it (*c* and *k* distinguish multiple objects of the same type). The backward-chaining metarule enables our intelligent assistant to automatically reserve modules whose components may have to be modified to restore consistency with the changed component. The metarule also prevents the user from modifying the specification of a component when dependent modules cannot be reserved (according to the first rule).

```
not reserved(module) and saved(module);
    { reserve module }
reserved(module, userid);

reserved(module, userid)
    { change component }
notanalyzed(component) and notcompiled(module);

for all components k such that in(module, component k)
    and uses(component k, component c):
        reserved(module, userid);
    { change component c }
```

Figure 2: Change Rules and Reserve Rule

3.3 Strategies and Hints

We chose the name 'opportunistic processing' for these chores because the assistant may perform an activity as the opportunity arises any time after its preconditions are satisfied and before another activity whose preconditions depend on its postconditions. Rules may be tagged so their activities are performed immediately after their preconditions are satisfied (*i.e.*, forward chaining applies) while other activities are performed only when their postconditions are required (forward chaining does not apply). Since we need to choose other points on this spectrum, we have included hints and strategies in our model to aid the intelligent assistant in making decisions.

A *hint* is similar to a rule, but without postconditions. The preconditions of a hint are used to guide the intelligent assistant in choosing when to apply a tool whose other preconditions are satisfied. Consider again the first rule from Figure 3.1. Suppose we do not want the assistant to compile a module, even though the preconditions are satisfied, while a user with modification rights is browsing through the module: The user may decide to change some components of the module, and the compilation will have been wasted. So we use a hint, Figure 3, giving this precondition for the "compile module" activity (angle brackets are used for parentheses). When the assistant follows a strategy including this hint, compilation is delayed until the user changes to another module.

```
not reserved(module) or
    < reserved(module, userid) and
      not equals(module, focus(userid)) >
    [ compile module ]
```

Figure 3: Compile Hint

Since we want the human user to be able to invoke the compiler without changing to another module, we give this precondition to "compile module" in a hint, rather than as part of a rule.

Hints apply only to the opportunistic processing of the intelligent assistant, not to activities initiated by a human user. In other words, hints are considered during forward chaining and ignored during backward chaining.

A *strategy* consists of a collection of hints and rules, which apply only when the strategy is in force. The third (and currently final) metarule from our model enables the intelligent assistant to employ strategies by combining its rules and hints with the rules normally considered. Zero or more strategies may be employed at the same time. When this results in more than one rule for the same activity, all their preconditions must be satisfied; only one set of postconditions is permitted.

Currently, our assistant cannot choose its own strategies; the knowledge to support this capability requires additional research on user modeling. Instead, each user will select appropriate strategies by informing the environment that he is, for example, a manager *vs.* a programmer, developing a new system *vs.* maintaining an old system, or making major changes *vs.* a minor revision. A strategy whose rules and hints result in automatically performing type checking immediately after each component is edited would be appropriate for a minor revision, but not for a large-scale changes involving many interrelated components.

3.4 Activities as Side-Effects

Often a tool performs additional activities as side effects. For example, the analysis tool invoked for the "analyze component" activity may change the values of several attributes of components. For the purposes of our rules, setting the value of an attribute is considered an activity, resulting in a situation where one action of the intelligent assistant is embedded inside another rather than being a consequence of forward or backward chaining. This case demonstrates a limitation of our rules: Secondary actions whose arguments cannot be determined in the general case cannot be expressed easily as postconditions. Instead, potential side effects are indicated by attributes of the tool.

In such cases, the secondary activities are often described by their own rules, and these must be considered for further processing. For example, some rules related to the "uses" attribute of a component are given in Figure 4. The "uses" attribute lists the components that the component depends on.

The first rule gives the obvious preconditions and postconditions for the "analyze component" activity. The second rule states a component *c* cannot use another component *k* unless *k* is in the same module or is imported into the module. The third rule means that a component cannot be imported by a module *M* unless it is exported by another module *N*. The fourth rule states that a component cannot be exported by a module unless it is in that module.

What happens when the analysis tool finds that procedure *p* (a component) calls procedure *q* (another component) and tries to set the "uses" attribute of procedure *p* to include procedure *q*? If *q* is in the same module as *p*, there is no problem; the attribute is set and the analysis con-

```
notanalyzed(component);
    { analyze component }
analyzed(component);
errors(component);

in(module, component c) and
    < in(module, component k) or imports(module, component k) >;
    { component c uses component k }
uses(component c, component k);

exports(module M, component) and
    not equal(module M, module N);
    { import component }
imports(module M, component);

in(module, component);
    { export component }
exports(module, component);
```

Figure 4: Analyze Rule, Uses Rule and Import/Export Rules

tinues. If *q* is not in the same module, the intelligent assistant checks whether it is imported. In the case where *q* is not already imported, the assistant notes that "imports(module, component)" is a postcondition of the "import component" activity (third rule) and realizes it can perform the "import component" activity without human intervention. It considers the preconditions of this activity. The assistant queries its objectbase to find the module that contains *q*. If *q* is already exported from that module, the assistant performs the "import component" activity. If not, the backward-chaining metarule permits the assistant to follow the preconditions of the activity given in the fourth rule of Figure 4. The assistant can add *q* to the exports of its module, then actually import *q* into the original module, and then permit the analysis tool to set the "uses" attribute of *p*.

In the above scenario, we ignored the possibility that distinct procedures named *q* might be found in more than one module. Sometimes language-specific typing information can be used to narrow down the possibilities, but generally the intelligent assistant must interrupt the human user to explain its dilemma and to ask which *q* is intended. The assistant can then proceed as described in the previous paragraph.

If no component named *q* is in the objectbase, the assistant considers the "add component q" activity, whose postcondition is the existence of *q*. A sufficiently intelligent assistant could carry out this activity by creating a stub for the procedure within the module where the use occurs; the Gnome programming environment for Karel does this automatically [10]. If this is not feasible, an alternative would be to ask the user to create the procedure (or stub) before continuing the analysis, but this would be intrusive; a preferred alternative is to inform the analysis tool of the problem and prevent it from performing the "procedure p uses procedure q" activity. This causes the analysis tool to terminate unsuccessfully, generating the "errors(p)" predicate among its postconditions.

In this discussion, "import component" and "export component" are among the activities that can be performed by the intelligent assistant without human intervention, permitting the assistant to carry out the repairs illustrated by the example. An alternative strategy would require the assistant to take the imports and exports as given. This might be appropriate for languages, such as Ada™, that include their own module constructs, where reference to an external component without the appropriate "with" clause should be detected as an error.

3.5 Implicit Queries

In the previous example, the assistant automatically queried its objectbase to locate procedure *q*. When the environment performs a query on its own, rather than in response to a user command, we call this an *implicit query*. Implicit queries are necessary to determine whether the preconditions of rules and hints are satisfied and to find the next rules to be applied in forward and backward chaining.

Another application is to anticipate the postconditions of activities, enabling the environment to warn the user when an action is likely to lead to adverse results. Consider again the two rules shown in Figure 5. Through forward chaining, changing a component will lead to semantic analysis, which may result in errors. When a user invokes the editor on a particular component, the environment anticipates this forward chaining and notes the possible "errors(component)" postcondition. This causes it to perform an implicit query to determine likely causes of the errors.

```
reserved(module, userid)
    { change component }
notanalyzed(component) and notcompiled(module);

notanalyzed(component);
    { analyze component }
analyzed(component);
errors(component);
```

Figure 5: Change and Analyze Rules

The intelligent assistant cannot guess what modifications the user will make and how these will affect other components. However, it can take advantage of the "used-by" attribute to determine those components most likely to be affected. Both the "used-by" attribute and its inverse ("uses") are listed in the objectbase among the potential side effects of the editor tool. The environment informs the user of potential sources of semantic inconsistencies by presenting the list of components given by the "used-by" attribute of the component argument to the editor. The user can take this information into account and choose whether or not to abort the "change component" command.

A further application of implicit queries was implied in Figure 2. A user gave the "change component" command, and backward chaining led the assistant to query the objectbase to deter-

mine whether all the modules affected by the proposed change were reserved by this user. If not, the environment would attempt to reserve all the necessary modules. However, this cannot succeed if some of these modules are reserved by other users. In this circumstance, the assistant presents the results of its implicit queries to the user to explain why the requested activity is not permitted.

3.6 Summary

The main points of our architecture for modeling the software development process are as follows.

- Rules define the preconditions that must be satisfied before a tool can be applied and the alternative postconditions of each tool.
- Hints define the preconditions that must be satisfied before a tool can be applied by the environment; unlike rules, hints do not affect the activities of human users.
- Two metarules define forward chaining from the postconditions of completed activities to the preconditions of other tools and backward chaining from the preconditions of desired activities to the postconditions of other tools.
- Tools may have side effects that cannot be expressed directly as postconditions, but these are nevertheless considered with respect to forward and backward chaining.
- The environment performs implicit queries to determine the attributes of software objects and the potential side effects of tools.
- Strategies group rules and hints appropriate for particular users and for particular phases of software development and maintenance. Our third metarule enables the intelligent assistant to consider these strategies during forward and backward chaining.

4 Implementation

We are implementing our intelligent assistant by reimplementing the internal mechanisms of SMILE. The advantages of starting with SMILE are (1) we can implement in place, retaining at all times a more-or-less working environment; (2) we have continued to use much of the previous SMILE code, most notably its disaster recovery mechanisms — approximately 30% of SMILE's source code protects against internal failures and supports recovery from external failures (disk full, system crashes, abort signals, *etc.*); and (3) we can continue to support the many systems, including SMILE, that have been developed and maintained using SMILE.

We are currently replacing SMILE's hardcoded knowledge about the software development process with the rules, hints and strategies of our model. The preconditions and postconditions of rules are translated into C routines that perform the corresponding queries and changes to the objectbase. The metarules are currently hardcoded. For example, forward chaining is performed by hashing on the actual postconditions of the most recently completed activity to find rules with potentially matching preconditions, which are then checked by the procedures that implement the preconditions.

We have so far retained SMILE's original objectbase, which is mapped onto the Unix file system, but we are currently designing a more flexible mapping that will support an extensible objectbase. We expect to complete this design and its implementation within three months. We have also retained the same user interface and tools, but expect to later replace the user interface to take advantage of bitmapped displays.

5 Conclusions

Our general architecture for intelligent assistance consists of an objectbase and a model of the software development process. The advantage of an objectbase is it permits the assistant to present a 'fileless environment' to its users, so the users are concerned only with the logical entities associated with software development and not with the details of the underlying file system and operating system. The advantages of a model of the software development process is that it can automate bookkeeping chores and other simple development activities and can constrain the invocation of tools to maintain consistency among the software objects.

These notions have been promoted by other researchers as the fundamental basis for a programming environment that understands what it does. The specific contribution of our research is the formalization of opportunistic processing, including implicit querying. Opportunistic processing is made possible by rules that describe the preconditions and postconditions of software development activities, metarules that permit the intelligent assistant to take advantage of these rules to enable automatic processing, and strategies that guide the assistant's application of the metarules. The result is a significant improvement in the assistance that the environment can provide for software development and maintenance by individuals as well as by teams of programmers, managers and other staff.

REFERENCES

[1] Robert Balzer.
A 15 Year Perspective on Automatic Programming.
IEEE Transactions on Software Engineering SE-11(11):1257-1268, November, 1985.

[2] Robert M. Balzer.
Living in the Next Generation Operating System.
In *Proceedings of the 10th World Computer Congress (IFIP Congress '86)*. Dublin, Ireland, September, 1986.
To appear.

[3] David R. Barstow, Howard E. Shrobe and Erik Sandewall.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[4] Lee Brownston, Robert Farrell, Elaine Kant and Nancy Martin.
Programming Expert Systems in OPS5.
Addison-Wesley Publishing Co., Reading, MA, 1985.

[5] CLF Project.
Introduction to the CLF Environment.
March, 1986.
USC Information Sciences Institute.

[6] Peter H. Feiler and Raul Medina-Mora.
An Incremental Programming Environment.
IEEE Transactions on Software Engineering SE-7(5):472-482, September, 1981.

[7] Peter H. Feiler and Gail E. Kaiser.
Intelligent Assistance without Artificial Intelligence.
Submitted for publication.

[8] S.I. Feldman.
Make — A Program for Maintaining Computer Programs.
Software — Practice & Experience 9(4):255-265, April, 1979.

[9] David B. Garlan and Philip L. Miller.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
In *Proceedings of the SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. Pittsburgh, PA, April, 1984.

[10] David Garlan.
Private communication.
July, 1986
Regarding capabilities of Gnome programming environments.

[11] Adele Goldberg.
The Influence of an Object-Oriented Language on the Programming Environment.
In *Proceedings of the 1983 ACM Computer Science Conference*. February, 1983.
Reprinted in [3].

[12] C.A.R. Hoare.
An Axiomatic Approach to Computer Programming.
Communications of the ACM 12(10):576-580, 583, October, 1969.

[13] Beverly L. Kedzierski.
Knowledge-Based Project Management and Communication Support in a System Development Environment.
In *Proceedings of the 4th Jerusalem Conference on Information Technology*. Jerusalem, Israel, May, 1984.

[14] David B. Leblang and Gordon D. McLean, Jr.
Configuration Management for Large-Scale Software Development Efforts.
In *GTE Workshop on Software Engineering Environments for Programming in the Large*, pages 122-127. June, 1985.

[15] John R. Nestor.
Toward a Persistent Object Base.
In *Proceedings of the IFIP WG 2.4 International Workshop on Advanced Programming Environments*. June, 1986.
To appear as a book published by Springer-Verlag.

[16] David Notkin.
The GANDALF Project.
The Journal of Systems and Software 5(2):91-105, May, 1985.

[17] L.J. Osterweil.
Toolpack — An experimental software development environment research project.
IEEE Transactions on Software Engineering SE-9(6), November, 1983.

[18] Dewayne E. Perry.
Position Paper: The Constructive Use of Module Interface Specifications.
In *Third International Workshop on Software Specification and Design.* London, England, August, 1985.

[19] C.V. Ramamoorthy, Vijay Garg and Rajeev Aggarwal.
Environment Modelling and Activity Management in Genesis.
In *Proceedings of SoftFairII: 2nd Conference on Software Development Tools, Techniques, and Alternatives*, pages 2-10. December, 1985.

[20] C.V. Ramamoorthy, Yutaka Usuda, Wei-TekTsai and Atul Prakash.
GENESIS: An Integrated Environment for Supporting Development and Evolution of Software.
In *The IEEE Computer Society's Ninth International Computer Software & Applications Conference (CompSac '85)*, pages 472-479. October, 1985.

[21] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems (TOPLAS) 5(3):449-477, July, 1983.

[22] M. J. Rochkind.
The Source Code Control System.
IEEE Transactions on Software Engineering SE-1:364-370, 1975.

[23] Douglas R. Smith, Gordon B. Kotik and Stephen J. Westfold.
Research on Knowledge-Based Software Environments at Kestrel Institute.
IEEE Transactions on Software Engineering SE-11(11):1278-1295, November, 1985.

[24] Barbara J. Staudt, Charles W. Krueger, A.N. Habermann and Vincenzo Ambriola.
The GANDALF System Reference Manuals.
Technical Report CMU-CS-86-130, Carnegie-Mellon University, Department of Computer Science, May, 1986.

[25] Mark Stefik and Daniel G. Bobrow.
Object-Oriented Programming: Themes and Variations.
AI Magazine 6(4):40-62, Winter, 1986.

[26] Tim Teitelbaum and Thomas Reps.
The Cornell Program Synthesizer: A Syntax-Directed Programming Environment.
Communications of the ACM 24(9), September, 1981.
Reprinted in [3].

[27] Warren Teitelman and Larry Masinter.
The Interlisp Programming Environment.
IEEE Computer 14(4):25-34, April, 1981.
Reprinted in [3].

[28] Walter F. Tichy.
RCS — A System for Version Control.
Software — Practice and Experience 15(7):637-654, July, 1985.

[29] Richard C. Waters.
KBEmacs: Where's the AI?
The AI Magazine VII(1):47-56, Spring, 1986.

[30] Terry Winograd.
Breaking the Complexity Barrier (Again).
In *Proceedings of the ACM SIGPLAN-SIGIR Interface Meeting on Programming Languages — Information Retrieval*, pages 13-30. Gaithersburg, MD, November, 1973.
Reprinted in [3].

END

7-87

DTIC